Programming Assignment #5: SneakyRooks

COP 3502, Fall 2018

Due: Sunday, December 2, *before* 11:59 PM

Abstract

This is a wrap-up assignment designed to reinforce the kinds of algorithmic and clever thinking we've been building up together over the course of the semester. In particular, this will serve as an additional exercise in coming up with efficient solutions to problems, since your solution for this assignment needs to have a worst-case runtime that does not exceed O($m + n$) (linear runtime). The assignment also involves a direct application of the base conversion material we covered recently (albeit with a minor twist).

You might find this problem a bit tricky at first. It's important to struggle with it. Don't be discouraged if you don't solve it right away. Maybe walk away, take a break, and come back to it later (perhaps even the following day). You might be amazed by what your brain can do if you let it work on a problem in the background and/or if you come back to a problem well-rested, with a fresh perspective.

Please feel free to seek out help in office hours if you're lost, and remember that it's okay to have conceptual discussions with other students about this problem, as long as you're not sharing code (or pseudocode, which is practically the same thing). Just keep in mind that you'll benefit more from this problem if you struggle with it a bit before discussing it with anyone else.

Deliverables

SneakyRooks.c

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

1. Problem Statement

You will be given a list of coordinate strings for rooks on an arbitrarily large square chess board, and you need to determine whether any of the rooks can attack one another in the given configuration.

In the game of chess, rooks can move any number of spaces horizontally or vertically (up, down, left, or right). For example, the rook on the following board (denoted with a letter 'R') can move to any position marked with an asterisk ('*'), and no other positions:

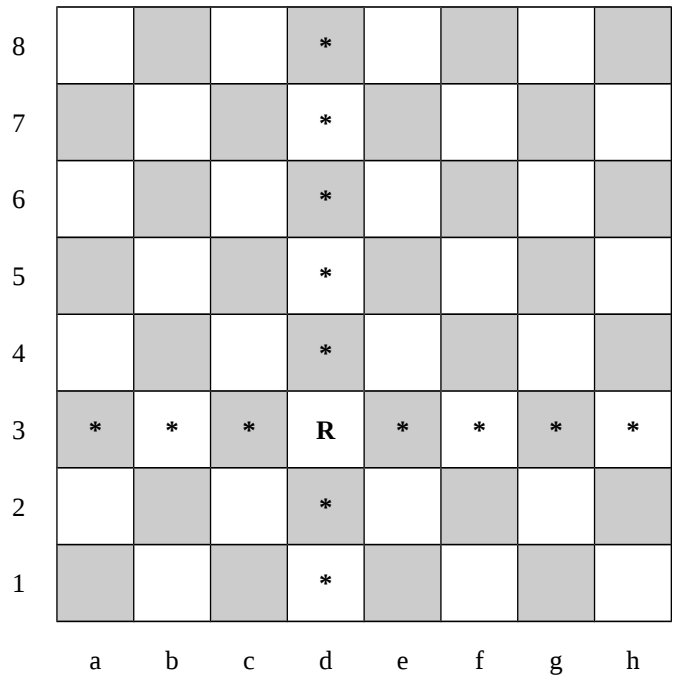

Figure 1: *The rook at position d3 can move to any square marked with an asterisk.*

Thus, on the following board, none of the rooks (denoted with the letter 'R') can attack one another:

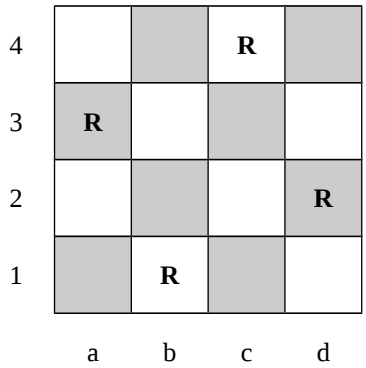

Figure 2: *A 4x4 board in which none of the rooks can attack one another.*

In contrast, on the following board, the rooks at *c6* and *h6* can attack one another:

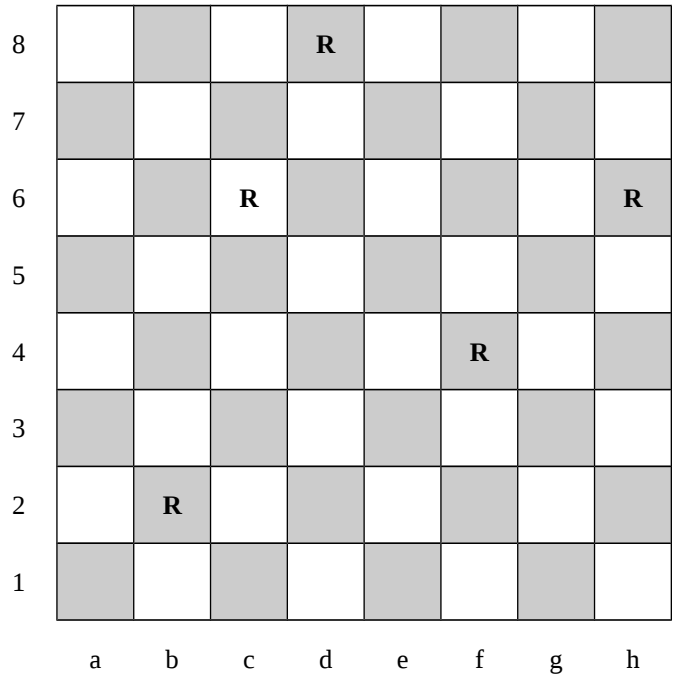

Figure 3: *An 8x8 board in which two of the rooks can attack one another.*

2. Chess Board Coordinates

2.1. Coordinate System

One standard notation for the location of a chess piece on an 8x8 board is to give its column, followed by its row, as a single string with no spaces. In this coordinate system, columns are labeled *a* through *h* (from left to right), and rows to be numbered 1 through 8 (from bottom to top).

So, for example, the board in Figure 2 (above, on pg. 2) has rooks at positions *a3*, *b1*, *c4*, and *d2*.

Because you're going to be dealing with much larger chess boards in this program, you'll need some sort of notation that allows you to deal with boards that have more than the 26 columns we can denote with the letters *a* through *z*. Here's how that will work:

Columns will be labeled *a* through *z* (from left to right). After column *z*, the next 26 columns will be labeled *aa* through *az*. After column *az*, the next 26 columns will be labeled *ba* through *bz*, and so on. After column *zz*, the next 26 columns will be labeled *aaa* through *aaz*.

Essentially, the columns are given in a base 26 numbering scheme, where digits 1 through 26 are represented using *a* through *z*. However, this counting system is a bit jacked up since there's no character to represent the value zero. (That's part of the fun.)

All the letters in these strings will be lowercase, and all the strings are guaranteed to be valid representations of board positions. They will not contain spaces or any other unexpected characters.

For example:

1. In the coordinate string *a1*, the *a* tells us the piece is in the first column (from the left), and the 1 tells us the piece is in the first row (from the bottom).

2. Similarly, the string *z32* denotes a piece in the 26th column (from the left) and 32nd row (from the bottom).

3. The string *aa19* represents a piece in the 27th column (from the left) and 19th row (from the bottom).

4. The string *fancy58339* would represent a piece in the 2,768,999th column (from the left) and the 58,339th row (from the bottom).

Converting these strings to their corresponding numeric coordinates is one of a few key algorithmic / mathemagical challenges you face in this assignment. You will have to write a function that does that for you.

2.2. Coordinate Struct (SneakyRooks.h)

To store rook coordinates, you must use the struct definition we have specified in `SneakyRooks.h` without any modifications. You must `#include` that header file from `SneakyRooks.c` like so:

```
#include "SneakyRooks.h"
```

The struct you will use to hold rook coordinates is defined in `SneakyRooks.h` as follows:

```
typedef struct Coordinate
{
   int col;  // The column where this rook is located (1 through board width).
   int row;  // The row where this rook is located (1 through board height).
} Coordinate;
```

3. Runtime Requirements

In order to pass all test cases, the worst-case runtime of your solution cannot exceed $O(m + n)$, where m is both the length and width of the square chess board, and n is the number of coordinate strings to be processed. This figure assumes that the length of each coordinate string is bounded by some constant, which means you needn't account for that length in your runtime analysis, provided that each string is processed or examined only some small, constant number of times (e.g., once or twice).

Equivalently, you may conceive of all the string lengths as being less than or equal to k, in which case the worst-case runtime that your solution cannot exceed would be expressed as $O(m + nk)$.

Note! $O(m + n)$ is just another way of writing $O(\text{MAX}\{m, n\})$, meaning that your runtime can be linear with respect to m or n – whichever one happens to be the dominant term for any individual test case.

4. Function Requirements

In the source file you submit, `SneakyRooks.c`, you must implement the following functions. You may implement any auxiliary functions you need to make these work, as well. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly.

Note: Your source file for this assignment **must not** contain a `main()` function.

```
int allTheRooksAreSafe(char **rookStrings, int numRooks, int boardSize);
```

> **Description:** Given an array of strings, each of which represents the location of a rook on a `boardSize` × `boardSize` chess board, return 1 if none of the rooks can attack one another. Otherwise, return 0. You must do this in O(`numRooks` + `boardSize`) time.
>
> **Parameter Restrictions:** `boardSize` will be a positive integer describing both the length and width of the square board. (So, if `boardSize` = 8, then we have an 8 × 8 board.) `rookStrings` will be a non-NULL (but possibly empty) array of strings, and any strings within that array will be unique (there will be no repeats), and all of those strings will follow the format described above for valid coordinates on a `boardSize` × `boardSize` board. `numRooks` will be a non-negative integer indicating the number of strings in the `rookStrings` array.
>
> **Output:** This function should **not** print anything to the screen.
>
> **Runtime Requirement:** The runtime for this function must be no worse than O(`numRooks` + `boardSize`). For details, see Section 3, "Runtime Requirements" (above).
>
> **Returns:** 1 if all the rooks are safe. Otherwise, return 0.

```
void parseCoordinateString(char *rookString, Coordinate *rookCoordinate);
```

> **Description:** Parse through `rookString` to determine the numeric row and column where the given rook resides on the chess board, and populate `rookCoordinate` with that information. You may assume that `rookString` is non-NULL, and that it contains a valid coordinate string using the format described above in Section 2.1, "Coordinate System." You may assume that `rookCoordinate` is non-NULL and is a pointer to an existing `Coordinate` struct.
>
> **Returns:** Nothing. This is a `void` function.

```
double difficultyRating(void);
```

> **Returns:** A double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

> **Returns:** An estimate (greater than zero) of the number of hours you spent on this assignment.

5. Special Requirement: Memory Leaks and Valgrind

Part of the credit for this assignment may be awarded based on your ability to implement the program without any memory leaks. To test for memory leaks, you can use a program called `valgrind`, which is installed on Eustis.

To run your program through `valgrind` at the command line, compile your code with the `-g` flag, and then run `valgrind`, like so:

```
gcc SneakyRooks.c testcase01.c -g
valgrind --leak-check=yes ./a.out
```

For help deciphering the output of `valgrind`, see: http://valgrind.org/docs/manual/quick-start.html

In the output of `valgrind`, the magic phrase you're looking for to indicate that you have no memory leaks is:

```
All heap blocks were freed – no leaks are possible
```

6. Test Cases and the test-all.sh Script

We've included a few test cases with this assignment to show some ways in which we might test your code and to shed light on the expected functionality of your code. We've also included a script, `test-all.sh`, that will compile and run all test cases for you.

The `test-all.sh` script is the safest, most sure-fire way to test your code.

You can run the script on Eustis by placing it in a directory with `SneakyRooks.c`, `SneakyRooks.h`, the `sample_output` directory, and all the test case files, and typing:

```
bash test-all.sh
```

Please note that we have only included a very small number of test cases this time around because we want to encourage you to create your own test cases and to think about how to test your code thoroughly. In creating your own test cases, you should always ask yourself, "How could these functions be called in ways that don't violate the function descriptions, but which haven't already been covered in the test cases included with the assignment?"

The fun continues on the following page!

7. Compilation and Testing (Linux/Mac Command Line)

To compile multiple source files (`.c` files) at the command line:

```
gcc SneakyRooks.c testcase01.c
```

By default, this will produce an executable file called `a.out`, which you can run by typing:

```
./a.out
```

If you want to name the executable file something else, use:

```
gcc SneakyRooks.c testcase01.c -o SneakyRooks.exe
```

...and then run the program using:

```
./SneakyRooks.exe
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following command, which will create a file called `whatever.txt` that contains the output from your program:

```
./SneakyRooks.exe > whatever.txt
```

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt sample_output/output01.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output:

```
seansz@eustis:~$ diff whatever.txt sample_output/output01.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt sample_output/output01.txt
1c1
< fail whale :(
---
> Hooray!
seansz@eustis:~$ _
```

8. Style Restrictions (*Super Important!*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

- ★ Any time you open a curly brace, that curly brace should start on a new line.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.
- ★ Please avoid block-style comments: /* *comment* */
- ★ Instead, please use inline-style comments: // *comment*
- ★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"
- ★ The header comments introducing your source file should always be placed above your #include statements.
- ★ Comments longer than three words should always be placed *above* the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.
- ★ Any libraries you *#include* should be listed *after* the header comment at the top of your file that includes your name, course number, semester, NID, and so on.
- ★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ When defining a function that doesn't take any arguments, always put *void* in its parentheses. For example, define a function using *int do_something(void)* instead of *int do_something()*.
- ★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*.
- ★ When defining or calling a function, do not leave a space before its opening parenthesis. For example: use *int main(void)* instead of *int main (void)*. Similarly, use *printf("...")* instead of *printf ("...")*.
- ★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.
- ★ Use meaningful variable names.

9. Special Restrictions

1. As always, you must avoid the use of global variables, mid-function variable declarations, and system calls (such as `system("pause")`).

2. Do not read from or write to any files. File I/O is forbidden in this assignment.

3. Be sure you don't write anything in `SneakyRooks.c` that conflicts with what's given in `SneakyRooks.h`. Namely, do not try to define a `Coordinate` struct in `SneakyRooks.c`, since your source file will already be importing the definition of a `Coordinate` struct from `SneakyRooks.h`.

4. No shenanigans. For example, if you write an `allTheRooksAreSafe()` function that always returns 1, you might not receive any credit for the test cases that it happens to pass.

5. Your `SneakyRooks.c` file **must not** include a `main()` function. If it does, your code will fail to compile during testing, and you will not receive credit for this assignment.

6. Be sure to include your name and NID as a comment at the top of your source file.

10. Deliverables

Submit a single source file, named `SneakyRooks.c`, via Webcourses. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work. Don't forget to `#include "SneakyRooks.h"` in your source code. Your program must compile on Eustis with both of the following:

```
gcc -c SneakyRooks.c
gcc SneakyRooks.c testcase01.c
```

Be sure to include your name and NID as a comment at the top of your source file.

11. Grading

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

100% passes test cases in linear time (see notes below)

Important Note! Additional point deductions may be imposed for poor commenting and whitespace. Significant point deductions may be imposed for violating the style restrictions or special restrictions listed above.

Note! Some of the test cases used in grading will check for memory leaks.

Note! Your program must be submitted via Webcourses, and it must compile and run on Eustis to

receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based primarily on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization, punctuation, or whitespace errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program thoroughly.

Start early. Work hard. Ask questions. Good luck!